practical : 01

Aim: Implementing the FCFS using C programming.

Theory:

- Meaning: The process that arrives first in the ready queue is executed first.
- Type: It is a non-preemptive scheduling algorithm (once a process starts execution it cannot be stoped until it finishes)
- Queue basis: processes are handled in the order of their arrival time just like customers waiting in line.

- characteristics of FCFS:

1. Simple to implement (like a queue data structure. - FIFO: Fir first in first out.

2. Fair Since every process is served in the order it arrives.

3. Non preemtive - a running process cannot be intrupted

Question: Solve this FCFS (First come first serve) CPU Scheduling problem Step by step

process P_2

- Arrival = 2
- Start = 8 (after P_1 finishes)
- completion = 8 + 4 = 12
- TAT = completion − Arrival = 12 − 2 = 10
- WT = TAT − Burst = 10 − 4 = 6

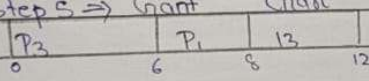
Step 3 ⇒ prepare Table

process	Arrival Time	Burst Time	Start	completion	TAT	WT
P_1	2	8	8	12	10	6
P_2	1	6	6	8	7	5
P_3	0	4	0	4	4	0

step 4 ⇒ Average Time

- Average TAT = (6 + 7 + 10) / 3 = 23/3 = 7.67
- Average WT = (0 + 5 + 6) / 3 = 11/3 = 3.67

Step 5 ⇒ Gant chart



final Answer

- Average waiting time = 3.67 ms
- Gantt chart = $P_3 \rightarrow P_1 \rightarrow P_2$
- Average Turnaround Time = 7.67 ms.

We are given:

process	Arrival Time	Burst Time
P_1	1 ms	2 ms
P_2	2 ms	4 ms
P_3	0 ms	6ms

Step 1 ⇒ Arrange processes by Arriving time.

- P_3 → Arrival 0
- P_1 → Arrival 1
- P_2 → Arrival 2

execution order : $P_3 \rightarrow P_1 \rightarrow P_2$ (Since FCFS is based on arrival)
6ms → 2ms → 4ms

Step 2: calculate start time, completion time, turnaround time (TAT) and waiting (WT)

process P_3

- Arrival = 0
- Start = 0
- Completion = 0+6 = 6
- TAT = completion - Arrival = 6-0 = 6
- WT = TAT - Burst = 6-6 = 0

process P_1

- Arrival = 1
- Start = 6 after P_3 finished
- completion = 6+2 = 8
- TAT = completion - Arrival = 8-1 = 7
- WT = TAT - Burst = 7-2 = 5

LINK:- https://www.programiz.com/online-compiler/0qxXnBCec5u14

CODE:_

```
#include <stdio.h>
int main()
{
   int n, bt[20], wt[20], tat[20];
   int avwt = 0, avtat = 0, i, j;

   printf("Enter total number of processes (maximum 20): ");
   scanf("%d", &n);

   printf("\nEnter Process Burst Time\n");
   for (i = 0; i < n; i++)
   {
      printf("P[%d]: ", i + 1);
      scanf("%d", &bt[i]);
   }
   wt[0] = 0;
   for (i = 1; i < n; i++)
   {
      wt[i] = 0;
      for (j = 0; j < i; j++)
         wt[i] += bt[j];
   }

   printf("\nProcess\t\tBurst Time\tWaiting Time\tTurnaround Time");
   for (i = 0; i < n; i++)
   {
      tat[i] = bt[i] + wt[i];
      avwt += wt[i];
```

```
    avtat += tat[i];
    printf("\nP[%d]\t\t%d\t\t%d\t\t%d", i + 1, bt[i], wt[i], tat[i]);
  }

  avwt /= n;
  avtat /= n;

  printf("\n\nAverage Waiting Time: %d", avwt);
  printf("\nAverage Turnaround Time: %d", avtat);

  return 0;
}
```

PRACTICAL 2

- implementing sjf using c programming

Aim: Implementing SJF using C Program

Definition:-
SJF is a Non-Preemtive scheduling algorithm where the process with the shortest burst time (execution time) is selected next for execution.

Note:- If two processes have the same burst time, then FCFS rule is apply for a tie-breaker.

Types:

1. Non-Preemtive SJF
- once a process starts a execution it cannot be stopped until it finished.

2. Preemtive SJF (also called as Shortest Remaining time first - SRTF)

- A process with a shorter remaining burst time can preempt the currently running process.

* Consider a following set of Processes with length of CPU Arrival time and burst time given in milliseconds.

Process	Arrival time	Burst time.
P1	1	3
P2	2	2
P3	0	5

illustrate the execution of these process using Non-Preemptive SJF Scheduling algorithm Calcute wait time, average wait time and turn around time, average turnaround time of each process is also draw the Gantt chart.

* Given Data:

Process	Arrival time	Burst time
P1	1	3
P2	2	2
P3	0	5

Step 1: Arrange Processes by arrival time. At time 0, only P3 has arrived → so the CPU start with P3 (SJF) is non-Preemptive, so once started, it finishes).

Step 2: Execution order.

- Time 0 → 5: P3 executes (Burst = 5). By time 5, P1 (arrival 1) and P2 (arrival 2) are already waiting.
- At time 5, choose the process with shortest burst among ready queue {P1, P2}
- P1 → 3, P2 → 2 → choose P2
- Time 5 → 7: P2 executes.
- Time 7 → 10: P1 executes.

final execution order: P3 → P2 → P1

Step 3: Completion Time (CT).

Process	Arrival	Burst	Completion Time (CT)
P3	0	5	5
P2	2	2	7
P1	1	3	10

Step 4: Turnaround Time (TAT)
Formula: TAT = CT - AT

Process	CT	AT	TAT
P3	5	0	5
P2	7	2	5
P1	10	1	9

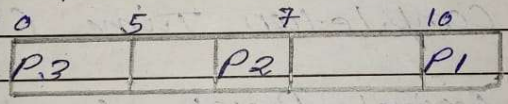
Step 5: Waiting time (WT)
Formula: WT = TAT − BT

Process	TAT	BT	WT
P3	5	5	0
P2	5	2	3
P1	9	3	6

Step 6: Averages.

- Average WT = (0+3+6)/3 = 3.0 ms
- Average TAT = (5+5+9)/3 = 6.33 ms

Step 7: Gantt Chart



Final Answer:

- Order of Execution: P3 → P2 → P1
- Average waiting Time = 3 ms
- Average Turnaround Time = 6.33 ms.

LINK:- https://www.programiz.com/online-compiler/5aYL9DqrCe561

Code:- #include <stdio.h>

```
int main() {
   int n, p[10], bt[10], at[10], wt[10], tat[10], temp;
   float wavg = 0, tavg = 0, tsum = 0, wsum = 0;
   int i, j, k = 1, btime = 0, min;

   printf("Enter the number of processes: ");
   scanf("%d", &n);

   for (i = 0; i < n; i++) {
      printf("Enter the burst time of process %d: ", i + 1);
      scanf("%d", &bt[i]);
      printf("Enter the arrival time of process %d: ", i + 1);
      scanf("%d", &at[i]);
      p[i] = i + 1;
   }


   for (i = 0; i < n - 1; i++) {
      for (j = i + 1; j < n; j++) {
         if (at[i] > at[j]) {
            temp = p[i]; p[i] = p[j]; p[j] = temp;
            temp = at[i]; at[i] = at[j]; at[j] = temp;
            temp = bt[i]; bt[i] = bt[j]; bt[j] = temp;
         }
      }
   }
```

```
for (j = 0; j < n; j++) {
   btime += bt[j];
   min = bt[k];
   for (i = k; i < n; i++) {
      if (btime >= at[i] && bt[i] < min) {
         temp = p[k]; p[k] = p[i]; p[i] = temp;
         temp = at[k]; at[k] = at[i]; at[i] = temp;
         temp = bt[k]; bt[k] = bt[i]; bt[i] = temp;
      }
   }
   k++;
}


wt[0] = 0;
for (i = 1; i < n; i++) {
   int total_bt = 0;
   for (j = 0; j < i; j++)
      total_bt += bt[j];
   wt[i] = total_bt - at[i];
   if (wt[i] < 0) wt[i] = 0;
   wsum += wt[i];
}

for (i = 0; i < n; i++) {
   tat[i] = bt[i] + wt[i];
   tsum += tat[i];
}

wavg = wsum / n;
tavg = tsum / n;
```

```
    printf("\nOUTPUT:-");
    printf("\nProcess\tBurst\tArrival\tWaiting\tTurnaround");
    for (i = 0; i < n; i++) {
        printf("\nP%d\t%d\t%d\t%d\t%d", p[i], bt[i], at[i], wt[i], tat[i]);
    }

    printf("\n\nAverage Waiting Time: %.2f", wavg);
    printf("\nAverage Turnaround Time: %.2f\n", tavg);

    return 0;
}
```

Practical 3

Implement producer-consumer synchronization using shared memory

and semaphores.

Link:- https://www.programiz.com/online-compiler/5gyIvUHmZuCds

Code :-

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
int count = 0;
int in = 0, out = 0;
```

```
pthread_mutex_t mutex;
pthread_cond_t notFull, notEmpty;

void* producer(void* arg) {
   for (int i = 1; i <= 10; i++) {
      pthread_mutex_lock(&mutex);

      while (count == BUFFER_SIZE) {
         pthread_cond_wait(&notFull, &mutex);
      }

      buffer[in] = i;
      printf("Producer produced: %d\n", i);
      in = (in + 1) % BUFFER_SIZE;
      count++;

      pthread_cond_signal(&notEmpty);
      pthread_mutex_unlock(&mutex);

      sleep(1);
   }
   return NULL;
}

void* consumer(void* arg) {
   int item;
   for (int i = 1; i <= 10; i++) {
      pthread_mutex_lock(&mutex);

      while (count == 0) {
         pthread_cond_wait(&notEmpty, &mutex);
```

```
        }

        item = buffer[out];
        printf("Consumer consumed: %d\n", item);
        out = (out + 1) % BUFFER_SIZE;
        count--;

        pthread_cond_signal(&notFull);
        pthread_mutex_unlock(&mutex);

        sleep(2);
    }
    return NULL;
}

int main() {
    pthread_t prodThread, consThread;

    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&notFull, NULL);
    pthread_cond_init(&notEmpty, NULL);

    pthread_create(&prodThread, NULL, producer, NULL);
    pthread_create(&consThread, NULL, consumer, NULL);

    pthread_join(prodThread, NULL);
    pthread_join(consThread, NULL);

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&notFull);
    pthread_cond_destroy(&notEmpty);
```

```
    return 0;
}
```